Practice Session 5: Design of python flask microservices and APIs

Make sure that you have already gone through Lab-04

Microservices is a variant of service-oriented architecture that consists of loosely coupled services composed in an application. It also enables the rapid, frequent and reliable delivery of large, complex applications. This lab aims to get acquainted with python flask microservices, create APIs using OpenAPI generator, and access using Swagger. Further, you will deploy this set of services using docker-compose.

The following tools/libraries are used in this lab to design microservices.

- **OpenAPI** : OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful APIs, which allows both humans and computers to discover and understand the service's capabilities without access to source code documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with minimal implementation logic. More information about OpenAPI
- **Swagger** : It's OAS specification to design the Restful APIs. Swagger allows you to describe the structure of your APIs so that machines can read them. The ability of APIs to describe their own structure is the root of all awesomeness in Swagger. We are using Swagger Editor to design the python APIs and validate using Swagger User interface More information about Swagger

In this experiment, we are going to perform the following tasks:

Note: We will be using CO2.CSV data in this experiment

1. Installing the Swagger Editor and Swagger user interface.
    a. Swagger Editor: We use this to design the micro services.
    b. Swagger User interface: We use this to test the micro services.
2. Create python flask based micro services to handle the IoT data. For this, we create methods using GET POST PUT and DELETE HTTP API operations.
3. Deploy the microservice application on Kubernetes cluster
4. Modify the application with extended operations such as *getMinimum* and *getMaximum* of CO2 values.

Exercise 1: Setting up of Swagger UI and Swagger Editor

The goal of this exercise is to get acquainted with Swagger components such as **Swagger Editor** and **Swagger UI**. Here, you will learn about YAML used to design the API documentation using OpenAPI specifications. In this task, you're going to set up the Swagger Editor using a docker container.

- Login to *k8s-controller* VM. You have created this VM in previous practice sessions.
- Install Docker compose
    - Download and save the executable file

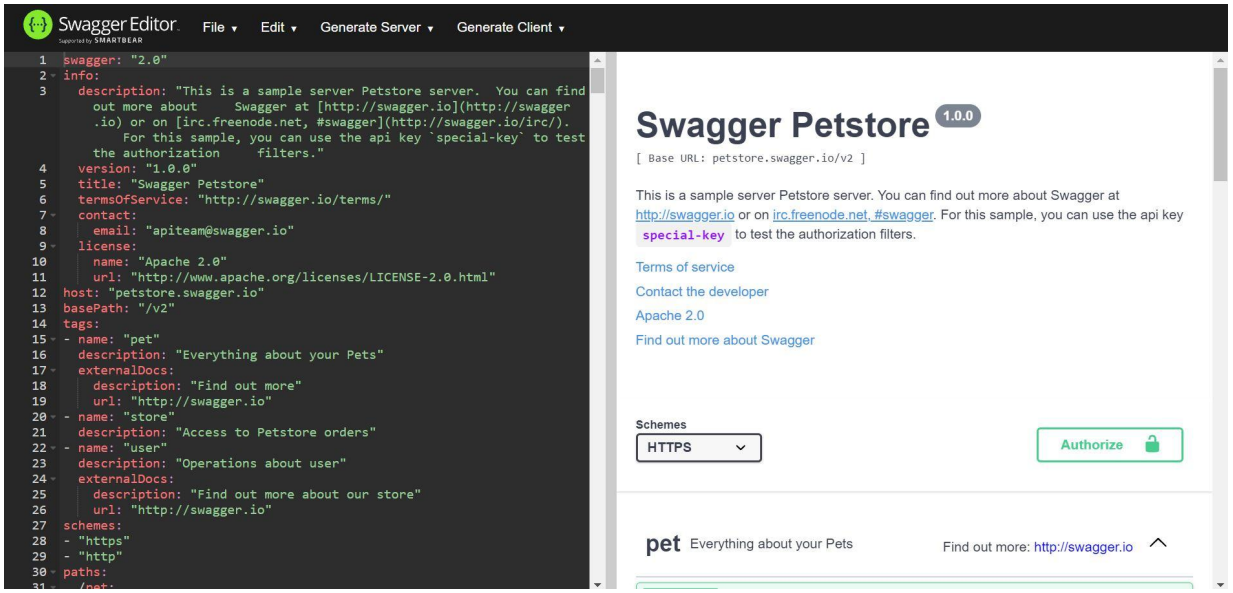
```
sudo curl -L
"https://github.com/docker/compose/releases/download/1.27.4/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

    - Change the mode of downloaded `docker-compose` file using `chmod` command: `sudo chmod +x /usr/local/bin/docker-compose`
    - Verify the installation of *docker-compose* using `docker-compose --version` command.
- Create a docker compose file `docker-compose.yaml` in your VM with following content:

```yaml
version: '3.3'
services:
  swagger-ui:
    image: swaggerapi/swagger-ui
    container_name: swagger-ui
    ports:
      - "8001:8080"
    volumes:
      - ./swagger:/usr/share/nginx/html/swagger
    environment:
      API_URL: swagger/api.yaml
  swagger-editor:
    image: swaggerapi/swagger-editor
    container_name: swagger-editor
    ports:
      - "8002:8080"
```

- Run the docker compose file `docker-compose up -d`
    - Make sure that the created `docker-compose.yaml` is present in the current path
    - Make sure that docker is running: `sudo systemctl status docker`
    - To run docker, if NOT running: `sudo systemctl start docker`
- Now open any web browser available in your laptop/PC and visit `http://k8s_CONTROLLER_VM_EXTERNAL_IP:8002`

- Now, you should be able to access the **Swagger Editor**, something similar to the below image:



- In the browser, on the left side you should see the editor, with some YAML content. This is the place to write your Swagger configuration file. (Recall the Swagger/OpenAPI configuration file from the lecture)
- Right side, you see the Swagger UI. Any modification to the YAML configuration will instantly reflect here. So, here also, you will instantly get errors if there is an error in the configuration editor.
    - Get handy with tabs in the editor
        - **File**: This helps you to create new yaml, import existing yaml and other operations.
        - **Edit**: This helps you to convert json to yaml and other operations such as convert from OpenAPI version 2 to version 3 document.
        - **Generate Server**: Once you finish designing the specification document, you can generate the template for your server-side code for various languages.
        - **Generate Client**: This helps to generate client-side code to invoke the APIs.
    - By default you will see an example of a `Petstore` server. So in the editor, you see the swagger configuration file of the `Petstore` app.
        - You may get acquainted with this example and explore this application.
        - Go through API documentation of `Petstore` application (right side of the **swagger editor** window).
    - Try to understand following keywords/keys in the YAML configuration file (left side of the window)
        - `swagger` version
        - `info`
        - `host` Your API endpoint address
        - `basePath` Your API version with base location of your API.
        - Learn about namespaces, tags, definitions from the YAML configuration document.(For information click here)

        - Learn the namespaces and corresponding methods.
- **Screenshot 1-1:** Take the screenshot of the Swagger editor web UI (should include the address bar of the browser)
- **Screenshot 1-2:** Execute the `docker ps` command in the terminal and take the screenshot

Exercise 2: Working with Swagger Editor and creating a python-flask microservice

This exercise aims to write your (REST API for python-flask microservice) OpenAPI/swagger configuration documentation using **Swagger Editor** and generate the corresponding server-side code.

- Open Swagger editor
- Select `File`-->`Clear Editor` and than start writing your own configuration
    - The first line should be the OpenAPI version. Lets use `swagger: "2.0"`
    - `info` - The info section contains API information: title, description (optional), version, etc.

```
info:
  description: "This is documentation for Python Flask micro-service
endpoints"
  version: "1.0.0"
  title: "Swagger IoT data"
  termsOfService: "http://swagger.io/terms/"
  contact:
    email: "xxx@ut.ee"
  license:
    name: "Apache 2.0"
    url: "http://www.apache.org/licenses/LICENSE-2.0.html"
```

- `host` indicates your server address where your microservice will run. You can use master node (node1) external IP address as the server address. It should look similar to `host: "NODE1_EXT_IP:8081"`
- `basePath` indicates the endpoint path. It can be `basePath: "/v2"`
- `schemes` indicate either http or https invocation, if https then need to provide authorization mechanism. In this exercise, we will use http. Add the element `http` under `schemes:`. It should look like below:

```
schemes:
- "http"
```

- paths are endpoints (resources), such as `/users` or `/reports/summary/`, that your API exposes, and operations are the HTTP methods used to manipulate these paths, such as `GET`, `POST` or `DELETE`.

Here we add POST operations under `/upload` namespace. The **tags** indicate that this operation belongs to `/upload` namespace. An operation definition includes parameters, request body (if any), possible response status codes (such as 200 OK or 404 Not Found) and response contents. (For more information)

```
paths:
  /upload:
    post:
      tags:
      - "Upload Sensor Data"
      summary: "Uploads a file."
      operationId: "upload_post"
      consumes:
      - "multipart/form-data"
      parameters:
      - name: "upfile"
        in: "formData"
        description: "The file to upload."
        required: false
        type: "file"
      responses:
        "200":
          description: "File uploaded"
        "400":
          description: "File not uploaded."

  /modify:
    put:
      tags:
      - "Modify Sensor Data"
      summary: "Updated Sensor name"
      operationId: "update_sensor_data"
      consumes:
      - "application/json"
      - "application/xml"
      produces:
      - "application/xml"
      - "application/json"
      parameters:
      - in: "body"
        name: "body"
        description: "device object that needs to be modified"
        required: true
        schema:
          $ref: "#/definitions/device"
      responses:
        "200":
          description: "Data modified"
        "400":
          description: "Data not modified"
```

- We continue the path with adding `/sensor_data` namespace that consumes a **host** as parameter and produces a json or string responses for different operations GET, DELETE.

```
/sensor_data/{host}:
   get:
     tags:
     - "Query Sensor Data"
     summary: "Find sensor data  by host"
     description: "Returns a list of sensor data"
     operationId: "get_sensor_by_id"
     produces:
     - "application/xml"
     - "application/json"
     parameters:
     - name: "host"
       in: "path"
       description: "Device id to search in the data"
       required: true
       type: "integer"
       format: "int64"
     responses:
       "200":
         description: "Successful operation"
         schema:
           $ref: "#/definitions/device"
       "400":
         description: "Invalid ID supplied"

   delete:
     tags:
     - "Delete Sensor Data"
     summary: "Delete sensor data"
     operationId: "delete_sensor_data"
     produces:
     - "application/json"
     parameters:
     - name: "host"
       in: "path"
       description: "Sensor Id that need to be updated"
       required: true
       type: "integer"
       format: "int64"
     responses:
       "200":
         description: "Successful operation"
       "400":
         description: "Invalid device id supplied"
```

- Likewise, we add the `/minimum{sensor}` and `/maximum{sensor}`, Over here, you will see a new key called **definitions** at the end. Try to search the purpose of this section.

```
/minimum/{sensor}:
   get:
     tags:
     - "Query Sensor Data"
     summary: "Find minimum sensor data  by host"
     description: "Returns a list of sensor data"
     operationId: "getminimum"
     produces:
```

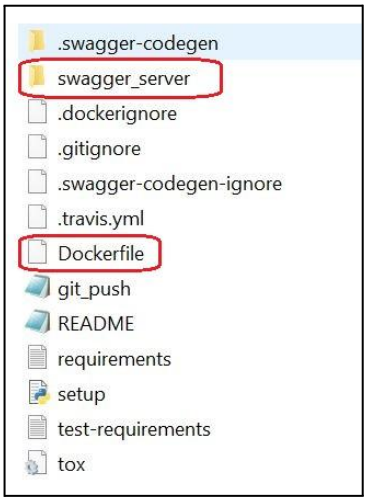
```
      - "application/xml"
      - "application/json"
      parameters:
      - name: "sensor"
        in: "path"
        description: "Host to search in the data"
        required: true
        type: "int64"
      responses:
        "200":
          description: "Successful with response containing minimum value"
          schema:
            $ref: "#/definitions/device"
        "400":
          description: "Invalid host supplied"

  /maximum/{sensor}:
    get:
      tags:
      - "Query Sensor Data"
      summary: "Find maximum sensor data  by host"
      description: "Returns a list of sensor data"
      operationId: "getmaximum"
      produces:
      - "application/xml"
      - "application/json"
      parameters:
      - name: "sensor"
        in: "path"
        description: "host to search in the data"
        required: true
        type: "int64"
      responses:
        "200":
          description: "Successful with response containing maximum value"
          schema:
            $ref: "#/definitions/device"
        "400":
          description: "Invalid ID supplied"
definitions:
  device:
    type: "object"
    properties:
      host:
        type: "integer"
        format: "int64"
      unit:
        type: "string"
        example: "kg"
    example:
      host: "13318"
      unit: "kg"
```

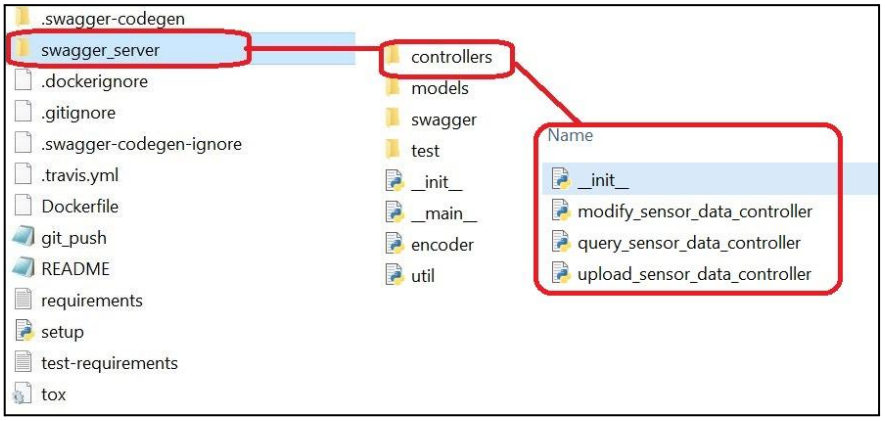
- The final configuration should look like this swagger.yaml.
- Check if you have any syntax errors in the document on the right side panel of the editor.
- Click on **Generate Server**-->**python-flask** and it will download the server-side code for you in compressed format.

- Extract the downloaded compressed file to your local machine (i.e. your laptop/PC) and you should have the following directory structure.

  Now you need to modify **swagger_server** directory and **DockerFile** file to implement the functionalities mentioned in the configuration file.



- You can find the configuration file inside **swagger_server** --> **swagger** --> **swagger.yaml**.
  - Open the Dockerfile and replace the first line from `FROM python:3-alpine` to `FROM amancevice/pandas:latest`. This is because we are using the pandas library in our further experiments with python 3.9.
  - We need to modify `upload_sensor_data_controller.py`, `modify_sensor_data_controller.py` and `query_sensor_data_controller.py` as shown in below figure.



- Now let us modify the controller code `upload_sensor_data_controller.py` as below.

  The logic/functionality is to receive the file(CO2.csv) invoked using `/upload` POST REST interface and save in to the local directory in `/tmp/iot.csv`

```
import connexion
import six
from flask import jsonify
from swagger_server import util
```

```
import pandas as pd
import json

def upload_post(upfile):  # noqa: E501

      df = pd.read_csv(upfile)
      df.to_csv("/tmp/iot.csv")
      data = {"message": "File Successfully uploaded"}

      return data, 200
```

- Likewise modify the `query_sensor_data_controller.py` with following code

```
import connexion
import six
from flask import jsonify

from swagger_server import util
import json
import pandas as pd


def get_sensor_by_id(host):  # noqa: E501

    try:
        df = pd.read_csv("/tmp/iot.csv")
        if host in df['host'].values :
            df_host = df.loc[df['host'] == host]
            data = df_host.to_json(orient="records")

            status = 200
            #data = {"Success message": "Device deleted from the csv"}
        else:
            status = 400
            message = {"Error message": "Invalid device"}
            data = json.dumps(message)
    except Exception as e:
            data = {"Error message": str(e)}
            status = 400
    return json.loads(data),status

def getmaximum(sensor):  # noqa: E501
    """Find maximum sensor data  by ID

    Returns a list of sensor data # noqa: E501

    :param sensor: Device id to search in the data
    :type sensor: str

    :rtype: Device
    """
    return 'do some magic!'


def getminimum(sensor):  # noqa: E501
    """Find minimum sensor data  by ID
```

```
    Returns a list of sensor data # noqa: E501

    :param sensor: Device id to search in the data
    :type sensor: str

    :rtype: Device
    """
    return 'do some magic!'
```

- Likewise modify the `modify_sensor_data_controller.py` with following code

```
import connexion
import six
from flask import jsonify

from swagger_server import util
import json
import pandas as pd


def update_sensor_data(body):  # noqa: E501
    """Updated Sensor name

    # noqa: E501

    :param body: Pet object that needs to be added to the store
    :type body: dict | bytes

    :rtype: None
    """

        # noqa: E501
    try:
        body = connexion.request.get_json()

        host = body['host']
        unit = body['unit']

        df = pd.read_csv("/tmp/iot.csv")
        if host in df['host'].values :
            df.loc[df['host'] == host, 'unit'] = unit
            status = 200
            data = {"Success message": "CSV updated and saved as
iot_updated_modifed.csv in /tmp"}
            df.to_csv("/tmp/iot.csv")
        else:
            status = 400
            data = {"Error message": "Invalid device"}
    except Exception as e:
            data = {"Error message": str(e)}
            status = 400

    return data, status
```

- Add the code for `delete_sensor_data_controller.py` with following code

```
import connexion
import six
from flask import jsonify
import json
import pandas as pd


def delete_sensor_data(host):  # noqa: E501
    """Delete sensor data

    This can only be done by the logged in user. # noqa: E501

    :param host: Sensor Id that need to be updated
    :type host: int

    :rtype: None
    """
    try:
        df = pd.read_csv("/tmp/iot.csv")
        if host in df['host'].values :
            indexNames = df[ df['host'] == host].index
            df.drop(indexNames , inplace=True)
            df.to_csv("/tmp/iot.csv")
            status = 200
            data = {"Success message": "Device deleted from the csv"}
        else:
            status = 400
            data = {"Error message": "Invalid device"}
    except Exception as e:
            data = {"Error message": str(e)}
            status = 400
    return jsonify(data),status
```

- Now update the `requirements.txt` file with following code:

```
connexion >= 2.6.0
connexion[swagger-ui] >= 2.6.0
python_dateutil == 2.6.0
setuptools >= 21.0.0
swagger-ui-bundle >= 0.0.2
flask_cors
```

- Now lets update the `swagger_server` --> `__main__.py` file with following code:

```
#!/usr/bin/env python3
```

```
import connexion
from flask import Flask
from swagger_server import encoder
from flask_cors import CORS

def main():
    app = connexion.App(__name__, specification_dir='./swagger/')
    app.app.json_encoder = encoder.JSONEncoder
    app_name = Flask(__name__)
    CORS(app_name)
    app.add_api('swagger.yaml', arguments={'title': 'Swagger IoT data'})
    app.run(port=8080)


if __name__ == '__main__':
    main()
```

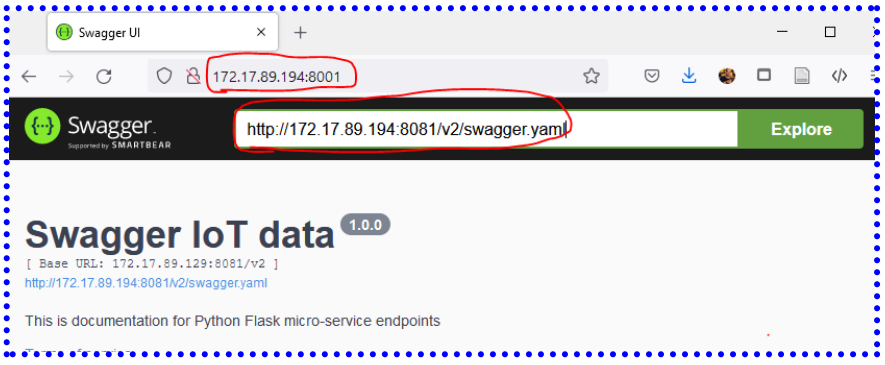
- At this point, the whole swagger project is in your local machine.
- Now, you need to push this to a new gitlab repository under your gitlab group .
- Create a project in the gitlab with name `lab05-flask-microservice` under group `Devops2022fall/students/devops2022Fall-<lastname>-<studyCode>`
- From the previous Lab, you know how to create a gitlab project locally and push to a remote Gitlab server. Follow the same and push the repository to your remote Gitlab group.

Exercise 3: Deploying your python-flask microservices

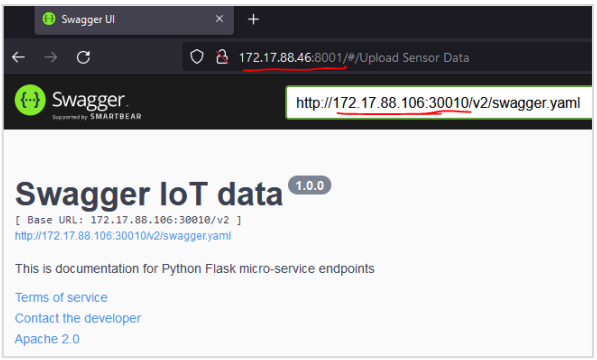
Here, the goal is to learn about deployment of the python-flask microservice in the kubernetes environment. For this, you need to build the docker file to create an image for your microservice and use the same image to run.

- Make sure you are in the terminal logged in to *k8s-controller* VM
- Building a micro service image:
    - Clone the project (that you have created in previous step) from GitLab using `git clone` command
    - Change the directory to your project containing the **Dockerfile**
    - Use build command `docker build -t <your_docker_hub_account>/flask-microservice .`
    - Push the docker image to your docker hub repository.
- Create and register a kubernetes agent. Refer Lab 04 - Task 4.2

    - This should be performed in master node (node1) VM.
    - Agent name could be: `microservice`
- Create a kubernetes deployment file `deployment.yaml`(You can use Web IDE, or create locally and push the code to gitlab)
    - This is similar to deployment created in the Kubernetes Lab 04.
    - Image name should be `<your_docker_hub_account>/flask-microservice`
    - Container port should be 8080
    - Add `hostPort` to 8081
- Add a block of kubernetes service in `deployment.yaml`
    - This is similar to deployment created in the Kubernetes Lab 04.
    - NodePort should be considered
    - Port 8080 should be considered
- Finally, to deploy a microservice to the kubernetes cluster, you have to create a `.gitlab-ci.yml`.
    - Contents of the file would be similar to tasks performed in the kubernetes lab.
    - Change the `kubectl use-context` to your created agent in the earlier step (Ex: `kubectl config use-context devops2022-fall/students/shiva-flask-microservice:microservice`).
    - Add the kubectl commands to create k8s deployment and services.
- Once committed the code, pipeline is executed and microservice is deployed.
- We test the microservice using **Swagger User Interface**
    - Visit http://K8s_CONTROLLER_VM_EXTERNAL_IP:8001/.
      Load the API definition as shown in the sample below.
      The Explorer textbox should look like below
      (http://k8s-node1_VM_EXTERNAL_IP:8081/v2/swagger.yaml)

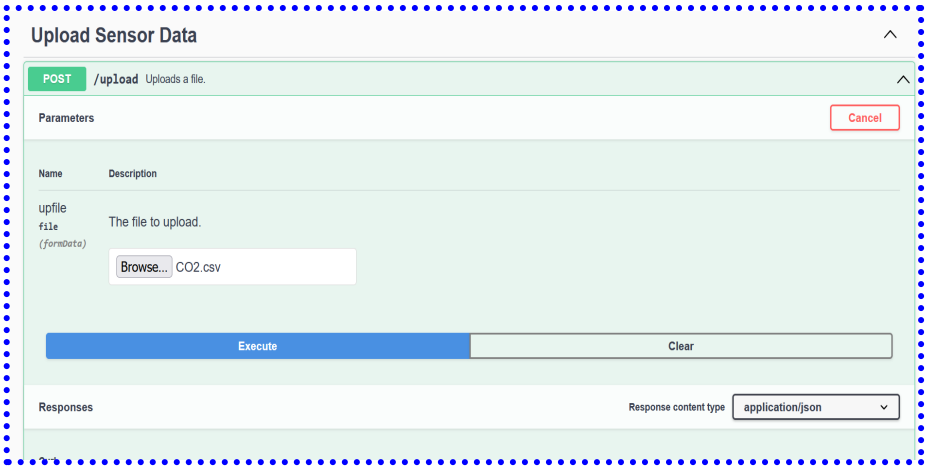


      UPDATE to above screenshot:
      Another configuration



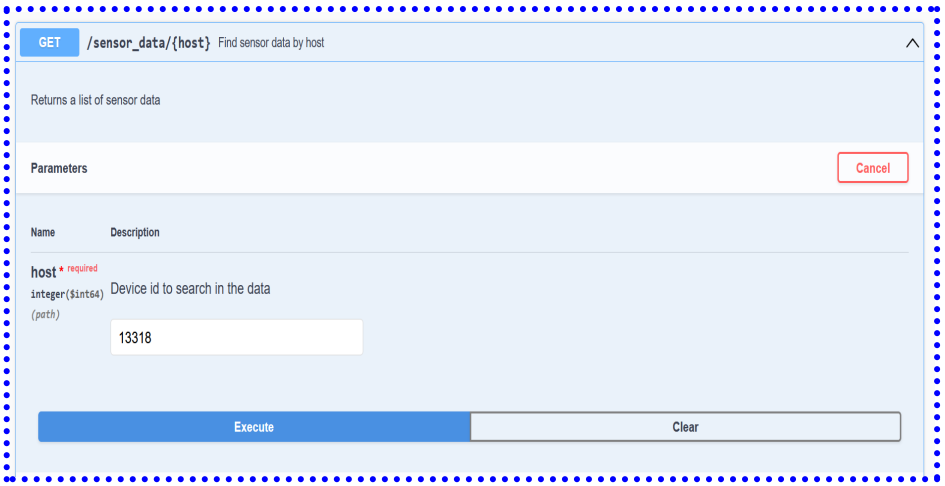
- 172.17.88.46:8001 is my controller VM and port number 8001 is where the Swagger UI is running
- http://172.17.88.106:30010/v2/swagger.yaml
  - this is my master node
  - 30010 is explicitly assigned nodePort in the deployment file.
  - 30010 is also mentioned in the swagger.yaml file. (host: "172.17.88.106:30010")
- To understand port, targetport, nodeport, please refer to this post (https://nigelpoulton.com/explained-kubernetes-service-ports/).

- **Screenshot 3-1:** take the screenshot of the web-page after the API definition is loaded. The address bar should be visible in the screenshot.
- Make sure you have downloaded the CO2.csv data from here CO2 data
- Now, let us test all the microservices endpoints of the python flask application using GET, POST,PUT and DELETE methods. For this you need to press the **Try Out** button at the right side of the page on every method.
  - POST : Upload the CO2.csv file to python-flask server using **POST** method



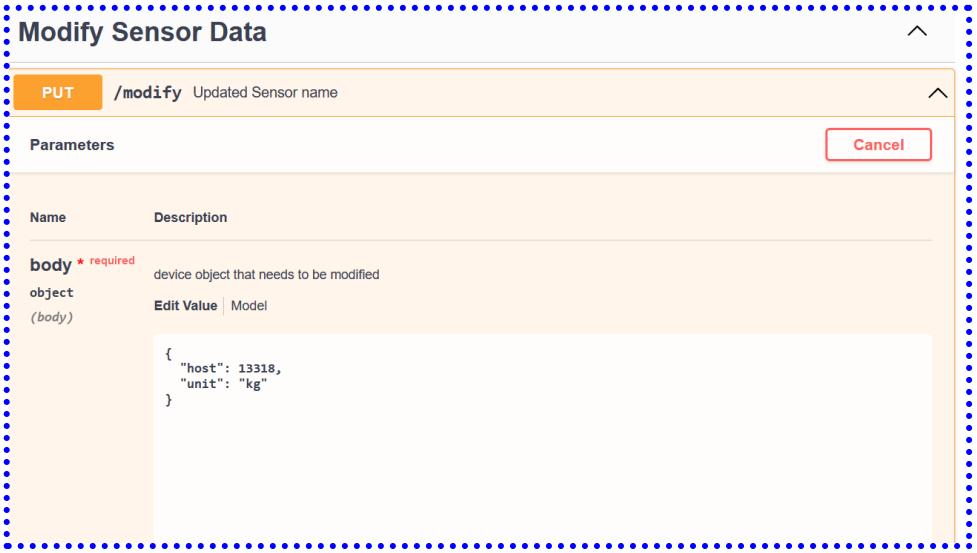
Screenshot 3-2: Take the screenshot of the response after executing **POST** method

- GET : Search for sensor data with host



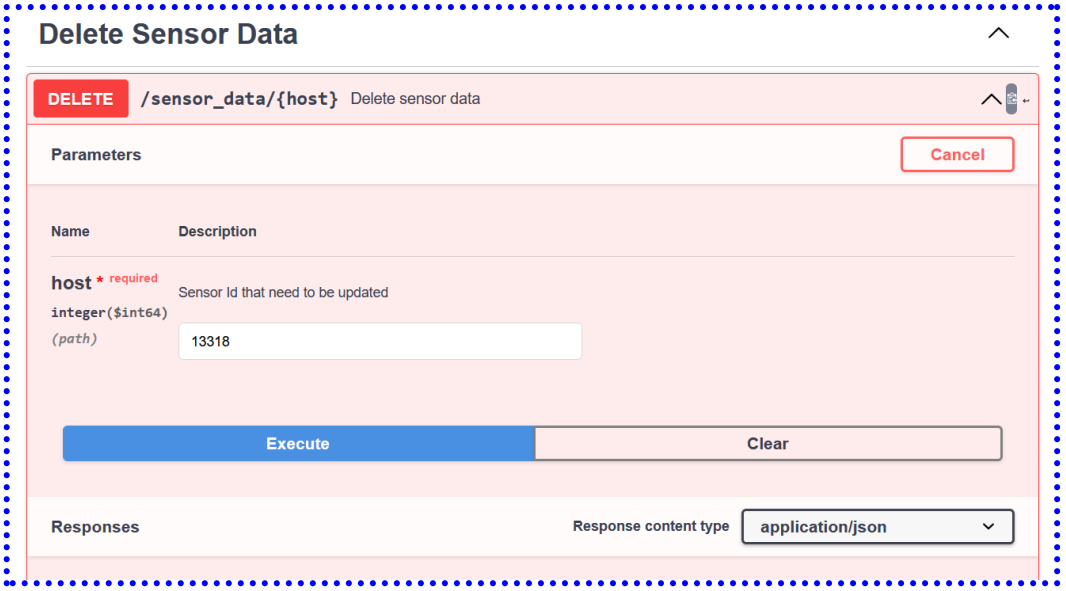
Screenshot 3-3: Take the screenshot of the response after executing **GET** method (address bar of the browser should be visible)

- PUT : Update the sensor name from given host



Screenshot 3-4: Take the screenshot of the response after executing **PUT** method (address bar of the browser should be visible)

- DELETE : Delete the sensors data from the csv for a given host



Screenshot 3-5: Take the screenshot of the response after executing **DELETE** method (address bar of the browser should be visible).

Exercise 4: Home Work : Additional python-flask micro services

In this task, you need to update the `query_sensor_data.py` file by writing the code for `getminimum` and `getmaximum`.

- In `getminimum`, you can use `dataframe.min()` function to obtain the minimum `value` of a given host.
- Likewise write the code `getmaximum` (Refer pandas guide for finding maximum column value in data frame)
- Build the image, deploy and test the application.
- Check the working of the methods using Swagger User Interface and please take the screenshots of the response of each method.

Screenshot 4-1: Take the screenshot of the response after executing **GET** method for `/minimum/{sensor}` namespace (address bar of the browser should be visible).

Screenshot 4-2: Take the screenshot of the response after executing **GET** method for `/maximum/{sensor}`(address bar of the browser should be visible)

Deliverable

1- Gather all the screenshots

- Screenshot 1-1
- Screenshot 1-2
- Screenshot 3-1
- Screenshot 3-2

- Screenshot 3-3
- Screenshot 3-4
- Screenshot 3-5
- Screenshot 4-1
- Screenshot 4-2

2- zip the swagger python-flask code, deployment files and all the screenshots.

3- Upload the zip file to the course wiki page.

4- You may **Stop** the Virtual Machines and you can start using the same in the next **practice session.**

Don't delete your VMs